Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Homework 9 due on 3/28
  - Assignment 2 due on 3/28
  - Lab 9 due on 4/1
  - Assignment 3 and 4 due on 4/18
    - Used to be one assignment

Previous lecture …

- Repeated Minimum Problem
  - Priority Queue and heap

CourseMIRROR Reflections (most confusing)

- graphic tracing
- The PQ sorting got confusing while tracing
- The heap sort was most confusing
- How to make the heap from the array was a bit confusing
- I was confused about when we would use a heap in an array or use it in a BTree structure. Are both done?
- Why would you use Prims algorithms versus Kruskals algorithm to find a MST?

CourseMIRROR Reflections (most interesting)

- Analysis of different type of tries
- Knowing the index of a child/parent of a node with the index formulas
- that you can represent a heap with just an array
- min heap insertion
- Seeing heaps and indexes. Some databases are built over these structures
- Heaps! And their find/insert/remove operations
- Kruskals seemed more straightforward than Prims
- I found it interesting how a HeapSort used the heap properties to sort effectively

Repetitive Minimum Problem

- Input:
  - a (large) dynamic set of data items in the form of
- Output:
  - find a minimum item
- You are implementing an algorithm that repeats this problem
  - examples of such an algorithm?
    - Prim's, Huffman tree construction
- What we cover today applies to the repetitive maximum problem as well

Let's create an ADT!

- The Priority Queue ADT

- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()
  - **Update an item**

Indirection

- Maintain a second data structure that maps item IDs to each item’s current position in the heap
- This creates an *indexable* PQ

Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

```
class CardPrice implements Comparable<CardPrice>{
    public String store;
    public double price;
    public CardPrice(String s, double p) { … }
    public int compareTo(CardPrice o) {
        if (price < o.price) { return -1; }
        else if (price > o.price) { return 1; }
        else { return 0; }
    }
}
```

Indirection example

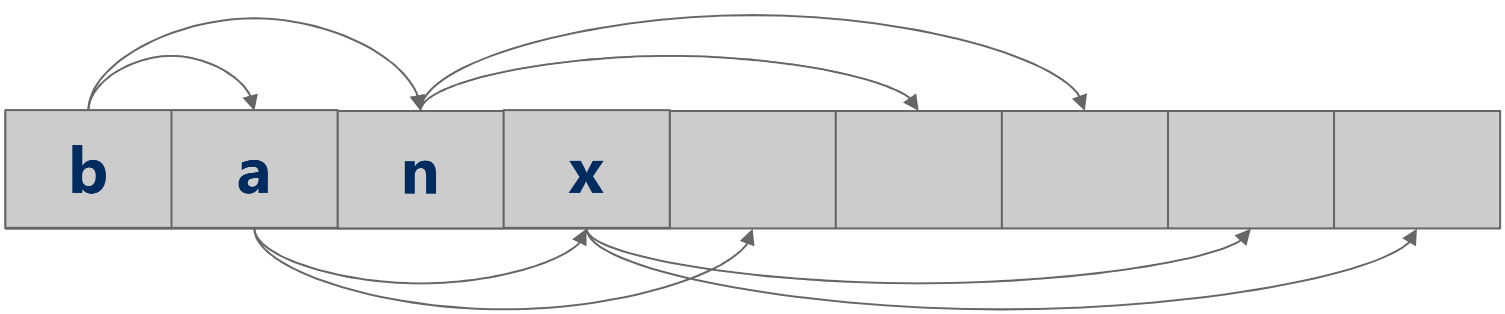
- n = new CardPrice("NE", 333.98);
- a = new CardPrice("AMZN", 339.99);
- x = new CardPrice("NCIX", 338.00);
- b = new CardPrice("BB", 349.99);

- Update price for NE: 340.00

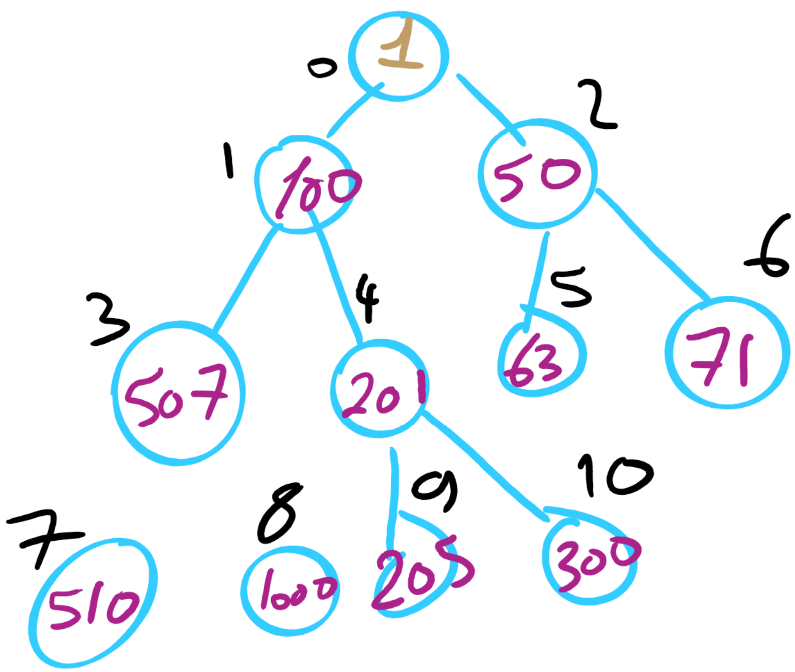
- Update price for NCIX: 345.00

- Update price for BB: 200.00

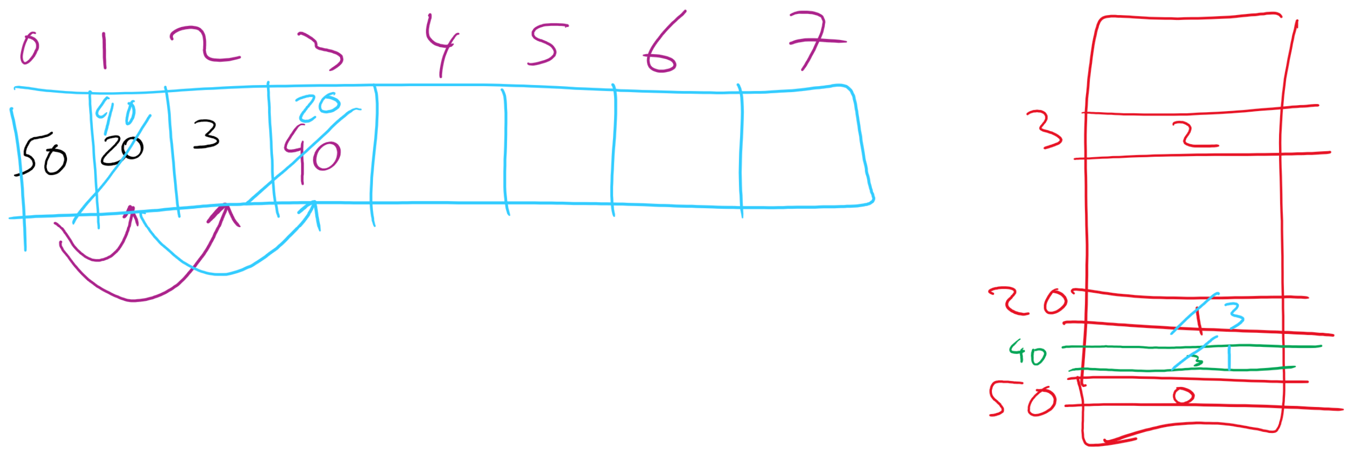
Indirection
"NE":2
"AMZN":1
"NCIX":3
"BB":0



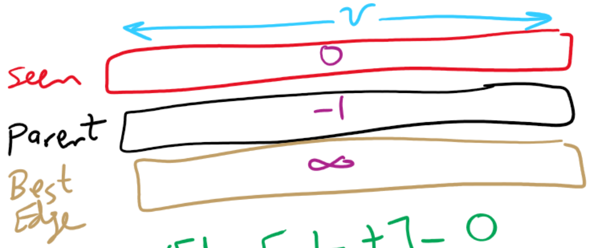
Indexable PQ Example



Indexable PQ Example



Prim’s MST Algorithm



BestEdge[start] = 0

for i = 0 → v-1

- find an unseen vertex w with the minimum BestEdge value $\Theta(v)$
- Seen[w] = 1
- for each neighbor x of w $\Theta(v)$
  - if (BestEdge[x] > edgeweight of (w, x)) $\Theta(1)$
    - BestEdge[x] = edgeweight of (w, x)
    - Parent[x] = w

$\Theta(v^2)$

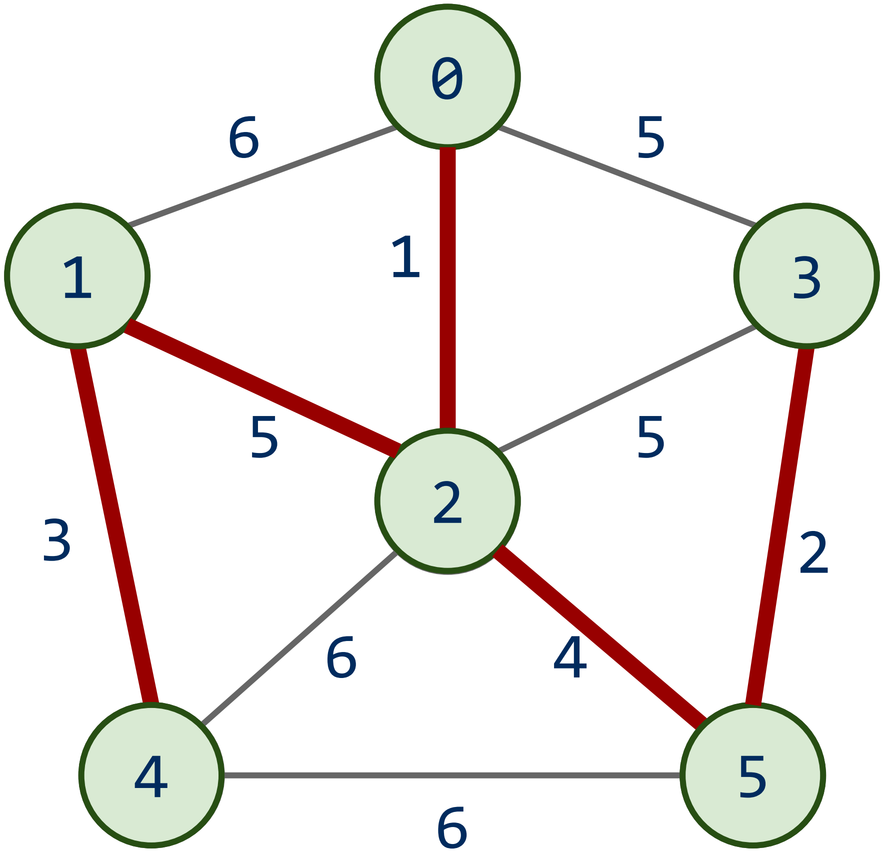
OK, so what's our runtime?

- For every vertex we add to T, we'll need to check all of its neighbors to check for edges to add to T next
  - Let's assume we use an adjacency matrix:
    - Takes $\Theta(v)$ to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
    - Time to pick next vertex?
  - What about with an adjacency list?

Prim's MST: What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in $\Theta(\lg n)$
    - Also $\Theta(\lg n)$ to add to the priority queue
- What does our algorithm look like now?
  - Visit a vertex
  - Add edges coming out of it to a PQ
  - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

1: (0, 2)

2: (5, 3)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (2, 1)

6: (0, 1)

6: (2, 4)

6: (5, 4)

Runtime using a priority queue

- Have to insert all e edges into the priority queue
  - In the worst case, we'll also have to remove all e edges
- So we have:
  - $e * \Theta(\lg e) + e * \Theta(\lg e)$
  - $= \Theta(2 * e \lg e)$
  - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Do we really need to maintain e items in the PQ?

- I suppose we could not be so lazy
- Just like with the adjacency matrix implementation, we only need the best edge for each vertex
  - PQ will need to be indexable
- This is the idea of *eager Prim's*
  - Runtime is $\Theta(e \lg v)$

Eager Prim's Runtime

v insertions : $v \log v$

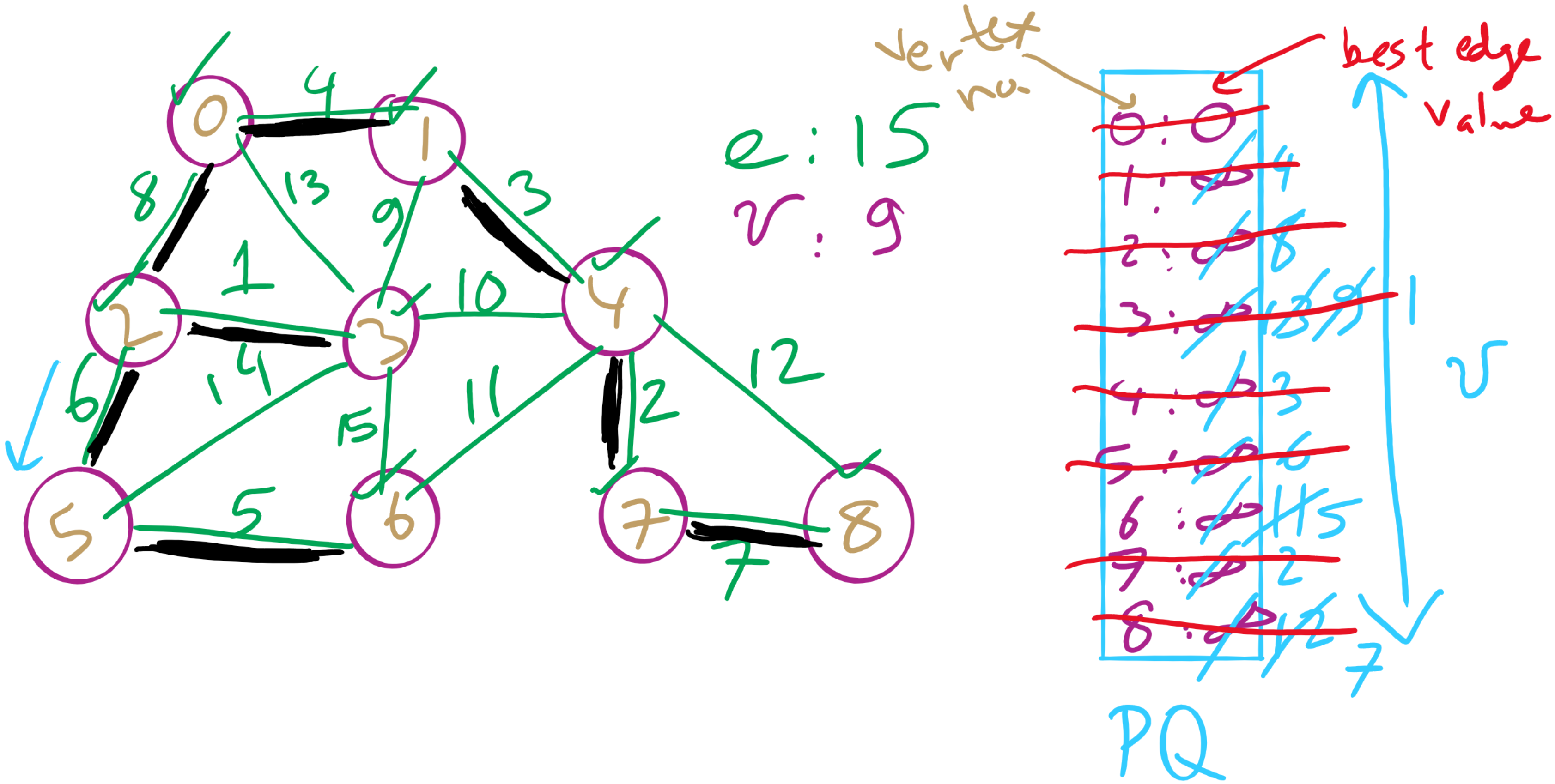
e updates : $e \log v$

v removals : $v \log v$

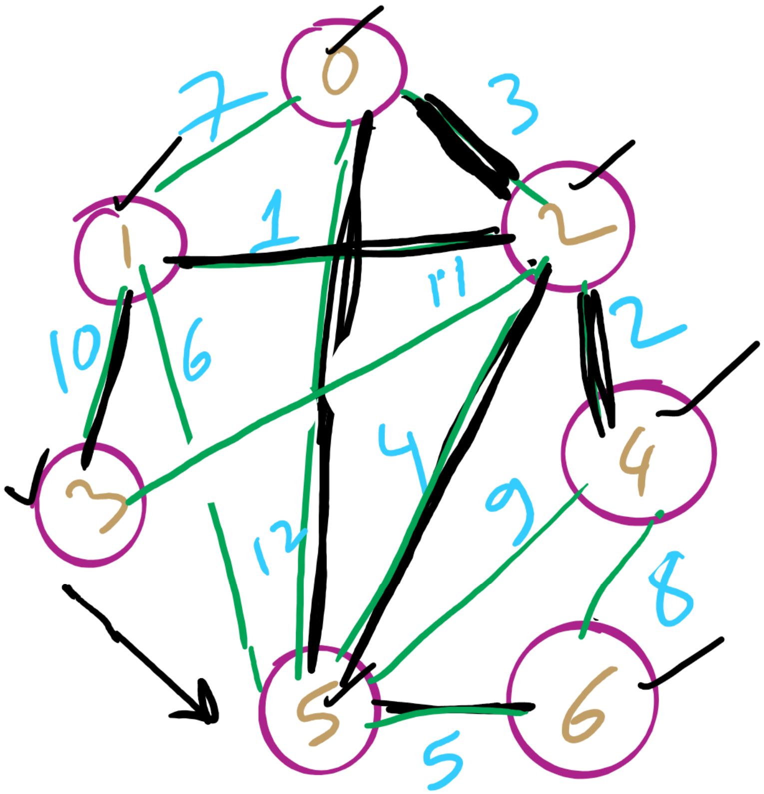
$$\overline{(e+v)\log v} = \Theta(e \log v)$$

$e \geq (v-1)$

Eager Prim's Example 1

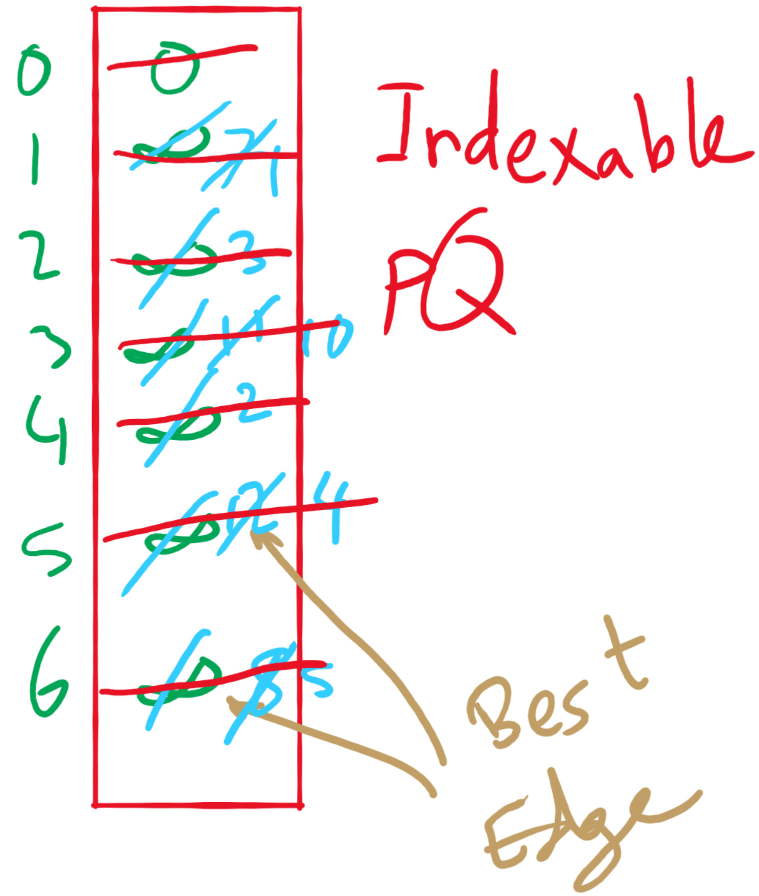


Eager Prim's Example 2



$v == 7$

$e == 12$



Comparison of Prim's implementations

- Parent/Best Edge array Prim's
  - Runtime: $\Theta(v^2)$
  - Space: $\Theta(v)$
- Lazy Prim's
  - Runtime: $\Theta(e \lg e)$
  - Space: $\Theta(e)$
  - Requires a PQ
- Eager Prim's
  - Runtime: $\Theta(e \lg v)$
  - Space: $\Theta(v)$
  - Requires an indexable PQ

How do these compare?

Eager vs. Lazy Prim's

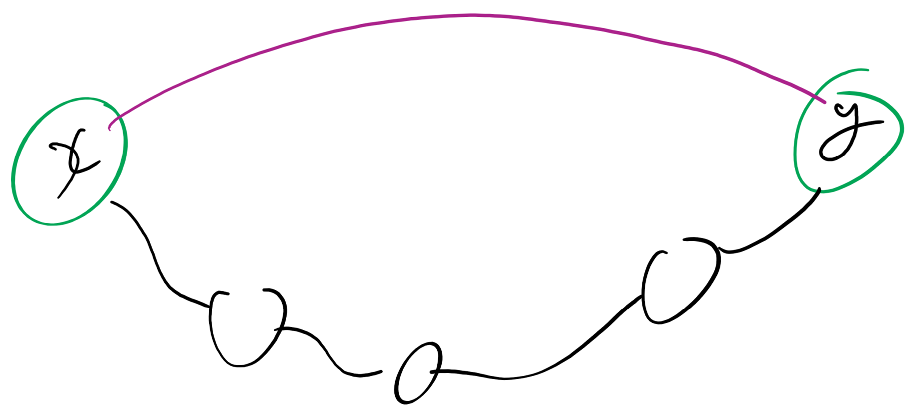
$$e \log e = e \log v^{2}$$

$$= 2 \cdot e \log v$$

$$= \Theta(e \log v)$$

Problem of the Day: Dynamic connectivity problem

- Input:
  - A set of items initially in separate groups and
  - a sequence of merge/union operations, each operation mering two items
- Output:
  - At any point of time, we can be asked if two items are in the same group
  - Initially, the answer will be NO for any two items because they start in separate groups
- For a given graph G, can we determine whether or
- Can also be viewed as checking subset membershi
- Important for many practical applications



Let’s build an ADT

- Union/Find ADT (aka Disjoint Sets ADT)

- Has two operations
  - Union(x, y)
    - Merge items x and y into the same group
  - Find(x)
    - Return the group number of x

A simple approach

- Have an *id* array simply store the component id for each item in the union/find structure
  - How do we determine if two vertices are connected?
  - How do we establish the connected components?
    - Add graph edges one at a time to UF data structure using *union* operations

Example

U(2, 0)

U(7, 4)

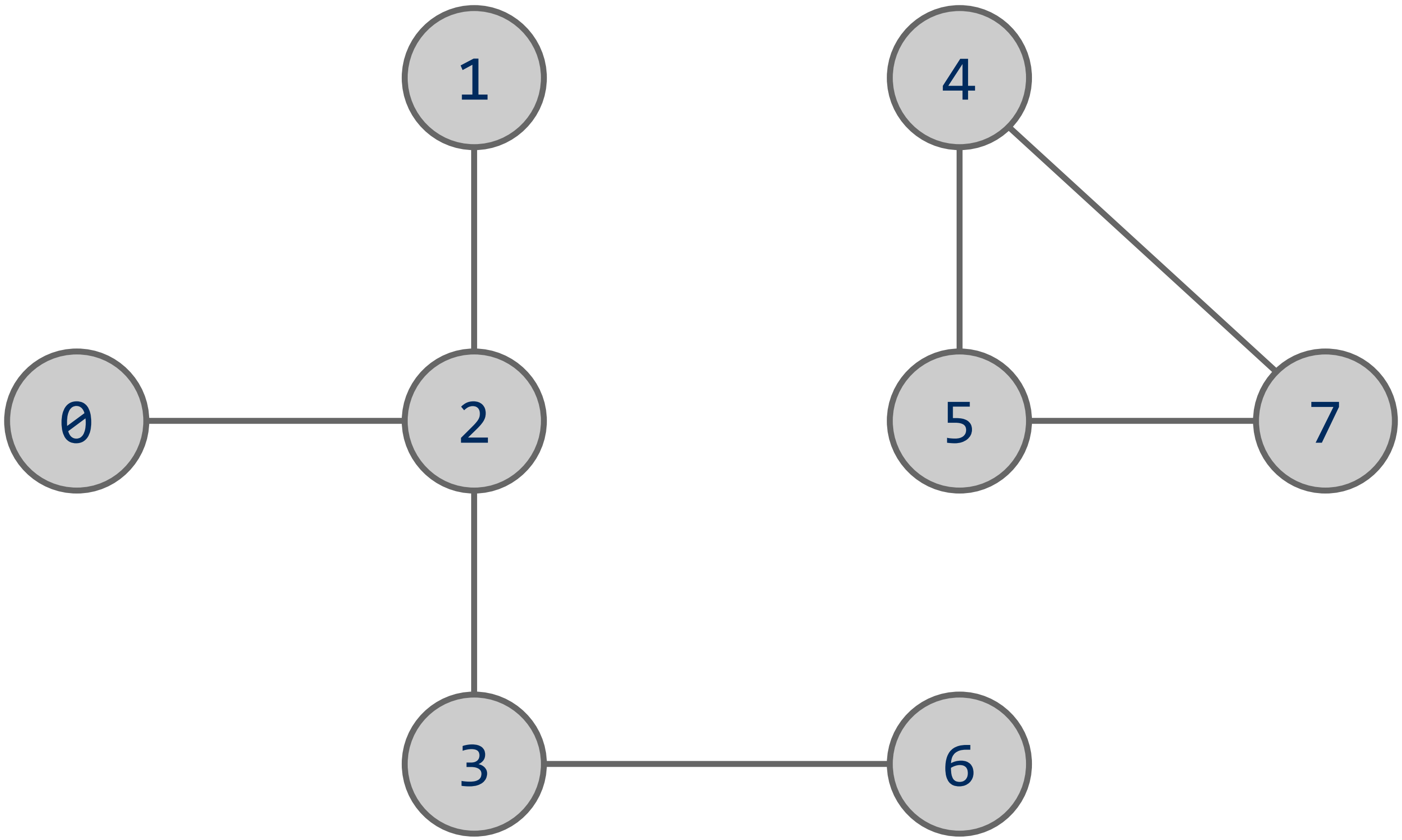
U(2, 1)

U(2, 3)

U(5, 4)

U(5, 7)

U(3, 6)



	0	1	2	3	4	5	6	7
ID:	6	6	6	6	4	4	6	4

Analysis of our simple approach

- Runtime?
  - To find if two vertices are connected?
  - For a union operation?

$\Theta(1)$ Connected?

$\Theta(n)$ union

Union Find API

Initialize with n items numbered 0 to n-1

Connect p with q

```
UF (int n)

void union(int p, int q)

int find (int p)

boolean connected (int p, int q)

int count()
```

Return id of the connected component that p is in

True if p and q are connected

Number of connected components

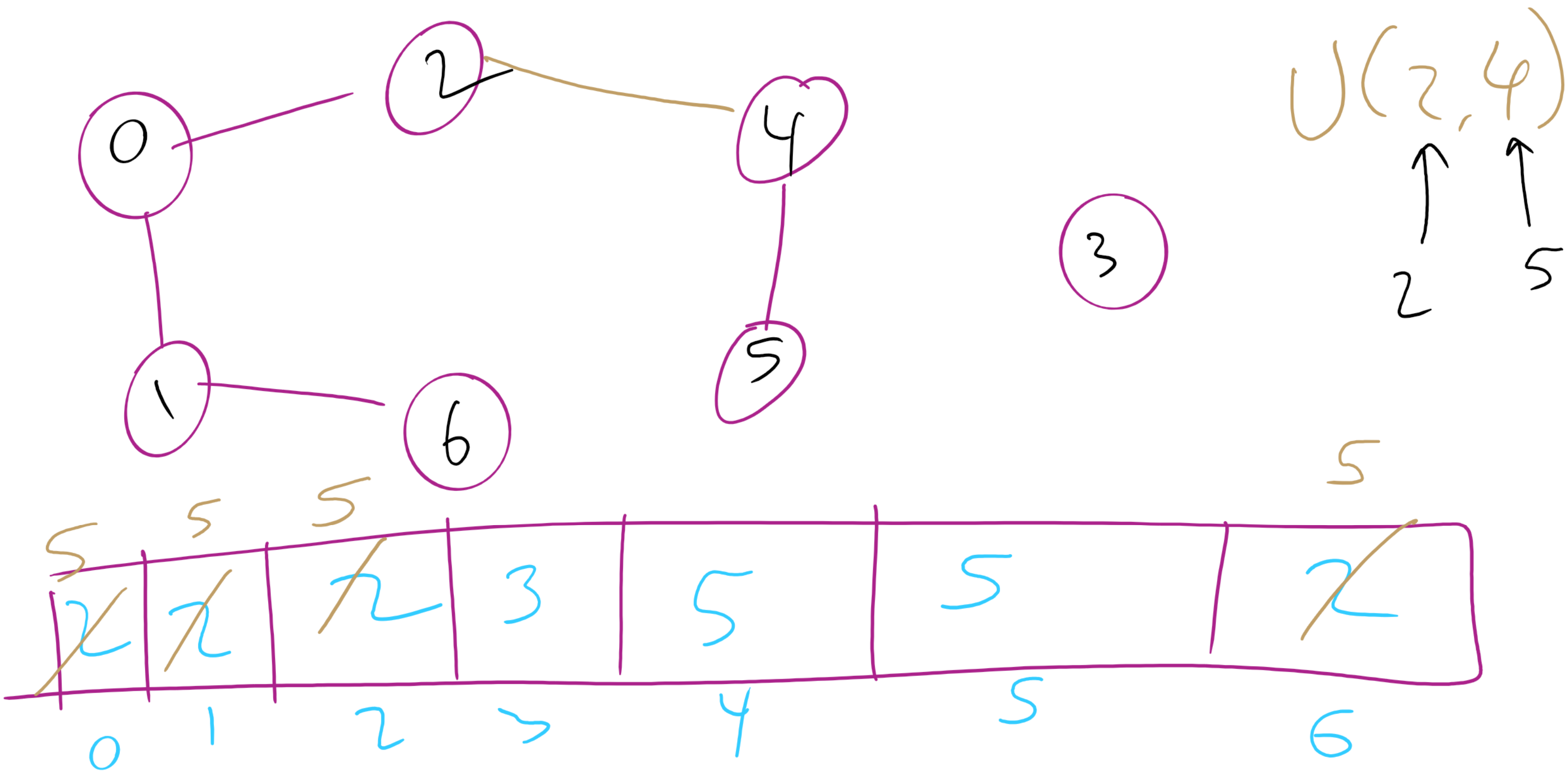
Covering the basics

```
public int count() {
    return count;
}

public boolean connected(int p, int q) {
    return find(p) == find(q);
}
```

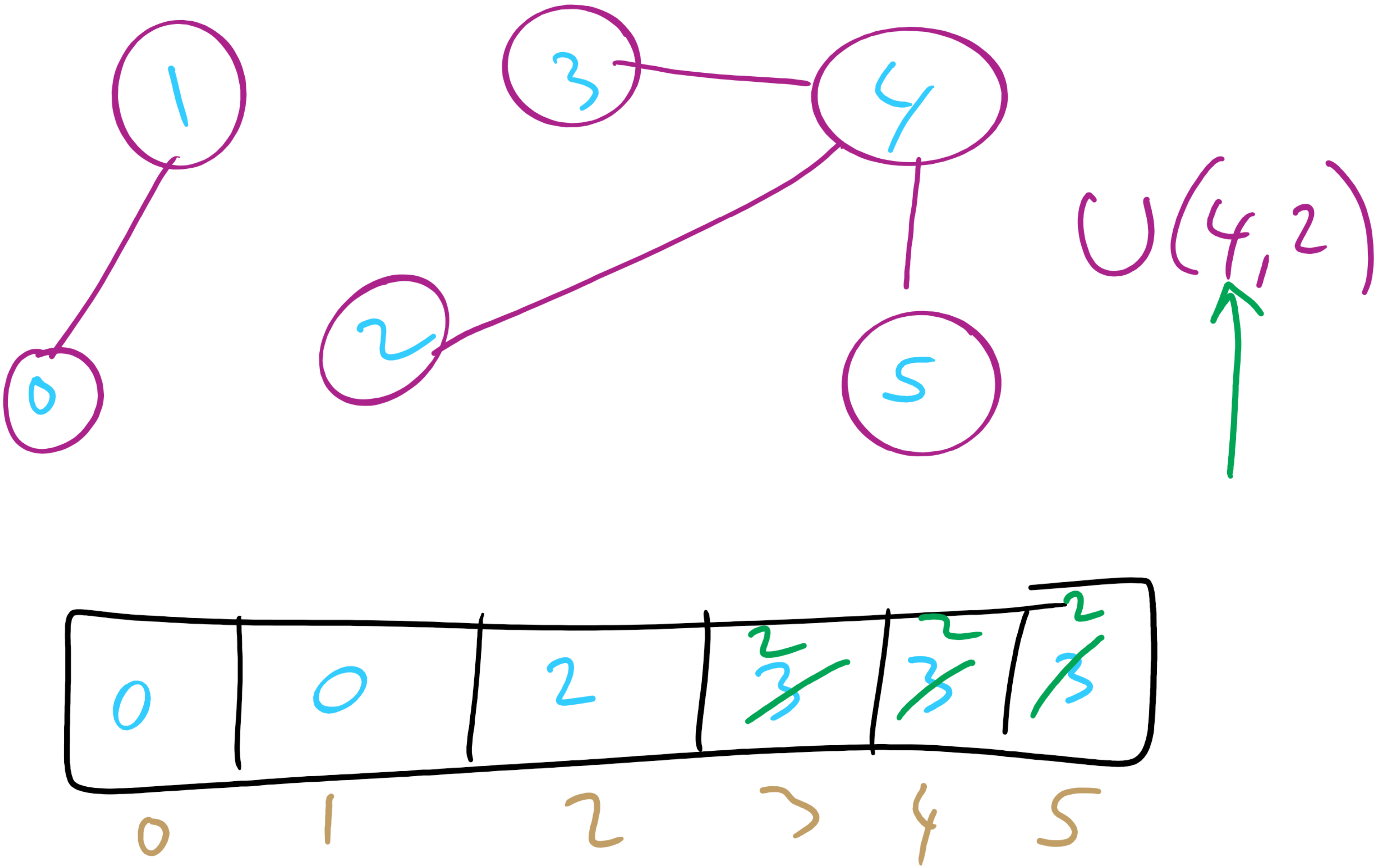
Implementing the Fast-Find approach

```
public UF(int n) {
    count = n;
    id = new int[n];
    for (int i = 0; i < n; i++) { id[i] = i; }
}

public int find(int p) { return id[p]; }

public void union(int p, int q) {
      int pID = find(p), qID = find(q);
      if (pID == qID) return;
      for(int i = 0; i < id.length; i++)
            if (id[i] == pID) id[i] = qID;
      count--;
}
```

Union-Find Example 1



Union-Find Example 2



Kruskal's algorithm Runtime: Take 2

- With this knowledge of union/find, how, exactly can it be used as a part of Kruskal's algorithm?
  - What is the runtime of Kruskal's algorithm?

e iterations

Connected? $\Theta(1)$

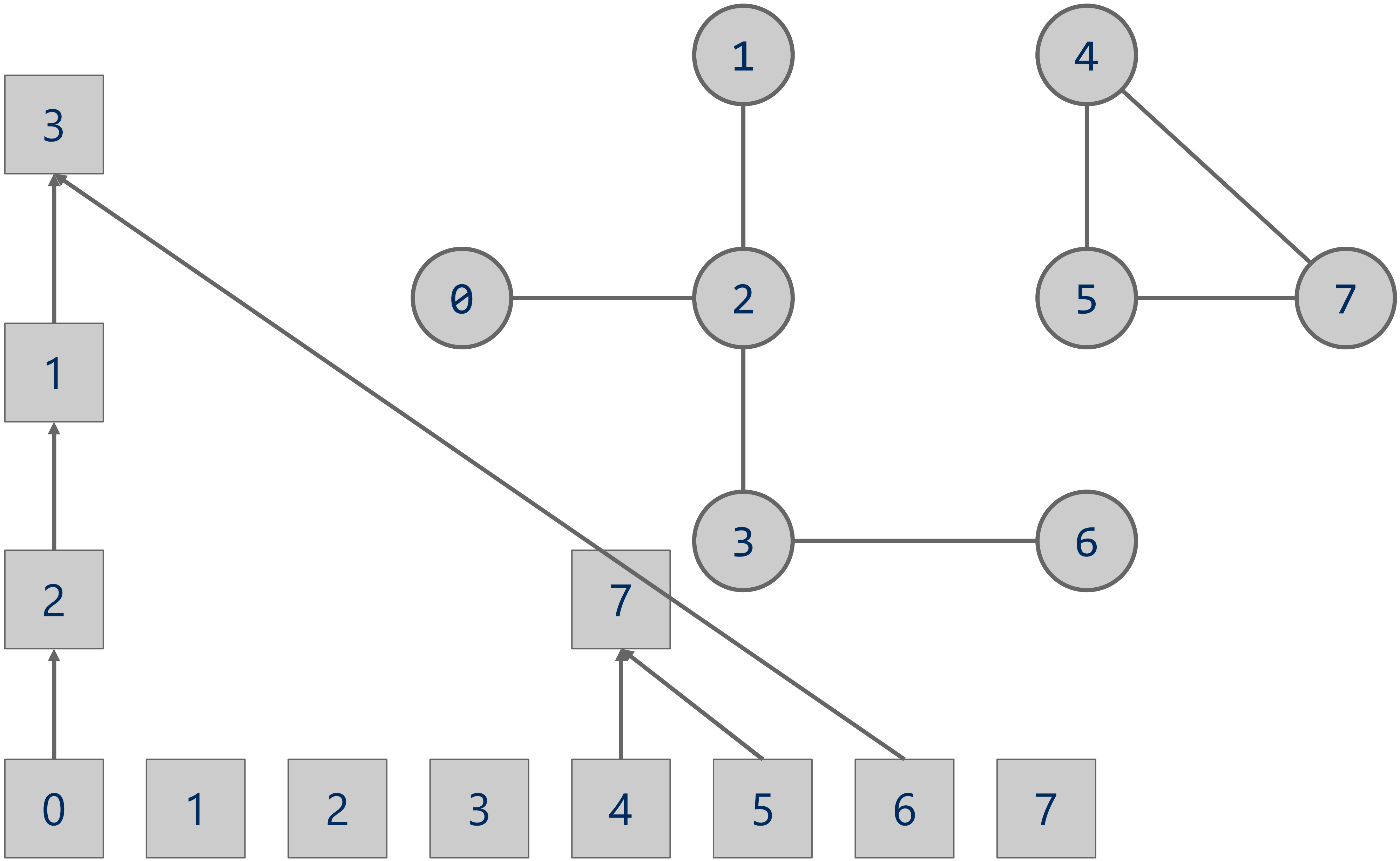
Union $\Theta(v)$

$\Theta(ev)$

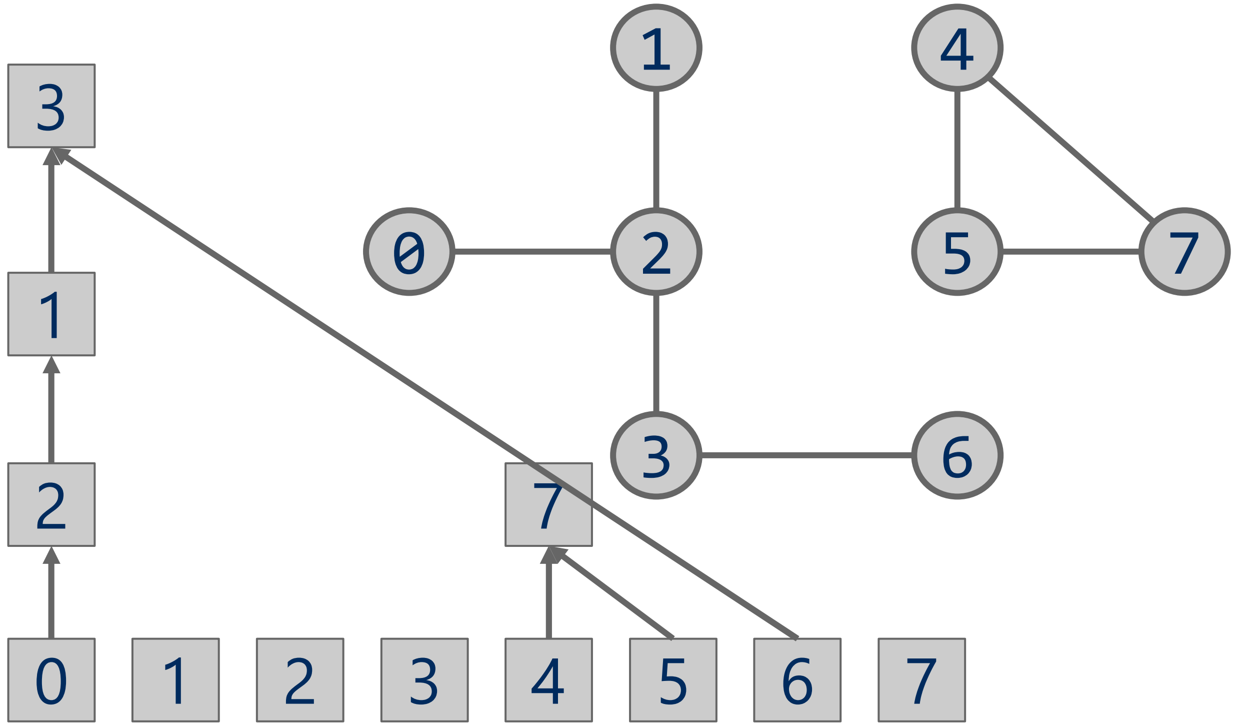
Can we improve on union()'s runtime?

- What if we store our connected components as a forest of trees?
  - Each tree representing a different connected component
  - Every time a new connection is made, we simply make one tree the child of another

Tree example



Forest of Trees Implementation



0	1	2	3	4	5	6	7
~~0~~ 2	~~1~~ 3	~~2~~ 1	3	~~4~~ 7	~~5~~ 7	~~6~~ 3	7

```
while( p != id[p])
    p = id[p];
```

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com